CHAPTER

3

DevOps Foundations

Culture, Lean Thinking, Metrics

Blink during a Formula 1 pit-stop and you'll probably miss it. But this wasn't always the case. Fifty years ago, a pit-crew would take over a minute to change the wheels and refuel. Today, anything more than three seconds is considered a fail.

It's the same in software development, where teams once tasked with updating enterprise applications at a sedate pace must now deliver new software services as a continuous flow of value to customers.

The problem for today's enterprise, however, is that software teams don't work like Formula 1 pit-crews. Rather than working in tandem, IT teams often work serially—development codes, then QA tests, and finally IT operations monitors. However, with application software released, enhanced, and retired over more compressed timeframes (months and even days), this stop-start method of development falls short. It's as ineffective as each member of a Formula 1 pit-crew replacing a tire and checking wheel nut tension before the next one could start—the race would be over before the car left the pits.

While we can celebrate the heroics and skill of great racing car drivers, what sets successful constructors apart is their ability to build a winning culture irrespective of role and responsibility, be that driver, team manager, telemetry engineer, or aerodynamics chief, everyone is focused on a singular goal—winning races. It's why drivers thank the teams before they spray champagne on the podium.

A. Ravichandran et al., *DevOps for Digital Leaders*, DOI 10.1007/978-1-4842-1842-6_3

Like Formula 1 drivers, technological advancements have improved the efficiency and effectiveness of IT professionals. However, in organizations that traditionally measure and incentivize based on technical specialization within functional areas, relying on tools alone will never build the collaborative culture needed for business growth and profitability.

What Characterizes DevOps Culture?

DevOps is very different from traditional thinking because it places great emphasis on culture. It instills a shared sense of vision across multiple teams, directly aligned to the business and its customers. To this end, maverick behavior, such a cutting corners and allowing defect ridden code to go into production, or blaming operations when a software release fails, is counter to a DevOps thinking. With DevOps unified IT is the hero and no one is singularly to blame for problems.

But this is challenging in IT because of the friction existing between development and other IT teams—especially IT operations. On the one hand, developers are focused on accelerating change by faster delivery of applications, while the operational mantra has been resilience and stability at all costs, even if that means holding back change.

Evidence suggests, however, that while both these goals are equally important, they are not mutually exclusive. For example, the 2016 Puppet Labs "State of DevOps" report illustrated that high-performing IT organizations are well able to achieve faster software delivery along with increased resilience and stability.[1] Clearly, DevOps high performers have ended the divisional "turf wars" by enacting strategies to re-shape entrenched silo thinking and behaviors into a more powerful collective force.

Since DevOps culture involves creating new shared values and behaviors across IT teams, leadership must play an active role in driving these characteristics across the entire organization.

Focusing on Products over Politics

Traditionally, IT teams have been organized in technical silos. Interaction and communication has been conducted through overly engineered and rigid processes. Software changes run the gauntlet of lengthy change-management processes, human intervention, and change review boards. Though not wrong per se, these elements were designed to cater to situations where change was less frequent but occurred in greater volumes, requiring more rigor to ensure operational stability and compliance.

[1] https://puppet.com/blog/2016-state-of-devops-survey-here

A strong DevOps culture, however, is characterized by systems thinking. That is, a collective emphasis on service as a whole, not on discreet functional elements or processes. Rather than persist with technical fiefdoms, DevOps aims to break down barriers—organizing by product over structure and continuously driving improvements in context of a products lifecycle, from the inception of an idea to full production status. Strong leaders recognize this by promoting open communication, using shared metrics, and establishing (even automating) feedback mechanisms within and across teams.

Building Trust and Respect

Over many years, respect has been garnered by individual contributors. Be that superhuman developers who crank out code, or on-call operations staff who fix problems at 4:00am. In a thriving DevOps culture, hero worshipping takes a back seat to collective respect. With DevOps, everyone should respect the contributions of others and no one should be afraid of speaking up for fear of abuse and vilification.

This is critical, because from healthcare to aerospace, studies have shown that bad practices and behaviors can over time become accepted as normal practices—often with disastrous consequences (see Chapter 8 for further discussion on strategies to combat normalized bad practices). In IT this happens all the time due to power games and lack of respect. Even if new staff witness blatantly suboptimal practices, they'll be loath to report it for fear of rebuke and retribution by managers, eventually accepting the situation and practicing it themselves. DevOps leaders should be mindful of this and work across the product lifecycle to identify situations where violations are tolerated because people are afraid to speak up or look mean.

Trust also plays an important role in DevOps culture. Just as the Formula 1 driver trusts his pit-crew to fit four wheels securely, so must cross-functional trust be established across IT. For development, this means trusting that the production performance information from operations can actually help in software refactoring and reducing technical debt. For operations, it means trusting new application design patterns will help the business scale. Everyone from security to enterprise architecture is part of the trust equation, and as the speed of software delivery accelerates, no DevOps program will be successful without it.

Increase Empathy Everywhere

It's well understood how important the role of empathy plays in today's app-centric software design. Without understanding the emotional and physical needs of customers, together with their behavioral patterns, businesses risk substantial losses from their software investments. This explains why many

organizations conduct rigorous design experiments before any full software release. This is illustrated in the extreme by Google's "50 Shades of Blue" user interface testing exercise.[2]

Yet despite this, empathy is lacking within many enterprise IT departments. Teams usually operate in separate locations, so development and operations teams have few face-to-face opportunities necessary to share each other's pains, surface concerns, or raise issues.

There are many simple but effective strategies leaders can use to build empathy. Not the least this should include building closer ties between development and support. Even with the greatest software and delivery processes it's important to understand their perspective and what they experience when dealing with customers.

Tip When developing products or new features, put yourself in the position of the customer and support staff by examining all the situations where they might need help.

Staff (including developers) should understand the importance of enabling a great customer experience. To that end, consider working directly with customers directly in the field in a variety of adverse situations. What happens when as a customer you're trying to board an airplane and the scanner breaks? Or what's the impact when a mobile app crashes or bad network coverage means you can't call road side assistance?

Obviously it's not always practical or feasible for developers to work this closely with clients; however, with analytics tools, staff can put themselves in the "shoes" of the customers and gain realistic insights into the customer experience.

Open Communication Channels

In 1968 a computer programmer, Melvin Conway, postulated that organizations that design systems are constrained to produce designs which are copies of the communication structures of these organizations.[3] In a DevOps context, what's now dubbed Conway's Law has great relevance, especially in situations where organizational structures and closed communication channels prevent developers and operations from agreeing on IT performance objectives (e.g., increased change frequency and improved reliability). In such cases, it's possible for team-based activities to be prioritized over more important cross-functional improvement strategies.

---

[2] https://www.theguardian.com/technology/2014/feb/05/why-google-engineers-designers

[3] Melvin E. Conway, "How do Committees Invent?," *Datamation* 14 (5) (April 1968): 28–31.

There are many possible solutions to this problem. The technology teams at Netflix and Amazon structure themselves around small teams, with each one responsible for a small part of an overall system. Spotify promotes collaboration across team boundaries via agile development squads, chapters, and guilds, with a separate IT operations team providing all teams the support needed to release software themselves.

Looking beyond technology-centric companies, there are other examples of businesses thriving because they've modified communication structures. Take Zara for example.

Year after year in the fickle world of retail fashion and apparel, Zara continues to increase revenue and profit. For Zara, flexible responsiveness to customer demand, backed by a tightly integrated supply chain, is fueled by teamwork and collaboration. In retail stores, managers use real-time intelligence to place orders and feedback information directly to the point of manufacture. Different teams (including design, product management, and merchandising) use shared spaces and work closely together. With increased emphasis on communication from initial garment design to distribution to the shop floor, Zara has shortened product lead times and doesn't unnecessarily commit large volumes of product in advance of a fashion season.

Considering approaches like these, cross-functional IT collaboration could be improved when:

- Leaders apportion budget to practical co-location strategies. This should be more than simple staff re-housing and include shared work spaces and lounges, together with team huddle areas and common wall whiteboards.
- IT operations team members regularly participate in agile standup meetings in order to appreciate the value of deploying code quickly and how their current activities help or hinder release processes.
- Development teams attend operational post-mortems or workshops to gain better insights into the problems caused by poorly performing or insecure software.
- IT operations works with developers to establish performance monitoring in pre-production so as to detect problems earlier where they are easier and less costly to fix.
- Developers are placed on the after-hours support roster to better appreciate the impact of problematic software code on users and customers.
- Support specialists share critical application experience analytics obtained during mobile app engagements with customers.

Additional Factors

Changing IT culture isn't easy. Right or wrong, people have pre-conceived notions and firmly entrenched ideas. Any sudden shift in workplace practices and it's only natural that people will feel threatened and push back. Addressing this means patiently working with people to enact the necessary behavioral change or move people to different jobs.

With DevOps and cultural change, it's critical to start with a clean state. Rather than dive head first into massive IT workforce transformation programs, leaders should first assess the cultural landscape from a business perspective. This involves understanding the primary goal of the business and then analyzing whether prevailing behaviors support it. Although seemingly obvious, many organizations miss or neglect this critical step. If, for example, a company defines its goals too broadly, people working in different IT teams will interpret them in different ways and shape activities accordingly, often to the detriment of each other and the business.

In a broader sense, culture will also be influenced by an organizations' business model and operational perspective. There are three classes to consider:

> *Run the business*—In this model the overarching strategy of the business is on continuing operations in much the same way—only better, faster, and cheaper. Here the focus for IT is operational excellence—adopting new technologies, yes, but using them to pound out efficiencies across processes like warehousing and logistics.
>
> For these organizations, IT culture is characterized by discipline and rigidity—all fine to support efficiency improvements, but inadequate in a world where old business rules are constantly being challenged by disruptive technology.
>
> *Grow the business*—The business strategy shifts from doing more of the same to conducting the same business in radically different ways. Netflix presents a good example of this model. Five years ago, Netflix delivered DVDs through the mail, now they stream entertainment over the web—even creating their own content. Customers still turn to Netflix for entertainment, but the way in which Netflix is addressing that need has fundamentally changed.
>
> For organizations in this category, the culture also needs to change. For Netflix, their DVD-delivery model required strong operational oversight over processes like inventory management and distribution to drive

efficiencies and increase customer satisfaction. Now however, their streaming model and content generation programs requires teams rapidly delivering new services based on quickly analyzing customer preferences and optimizing web performance and throughput. Obviously if a company's cultural behaviors and values are still skewed toward driving efficiency in one operational area, pivoting to the new model will be difficult.

Transform the business—This model carries the most promise and risk because it involves changing the very fabric of a company. Businesses don't just conduct the same types of business in new ways, they reinvent themselves completely. For Amazon that's meant moving to being a mega cloud computing provider from selling books. For Walgreens, it's meant going from selling medicine over the counter to treating illnesses in stores.

With strategic transformation examples like these, the business is introduced to new dynamics and competitors. Now, IT performance will not only be judged on growing the business of today, but also on creating the core business in the future. To this end, a DevOps culture built on open communication and collaboration, trust, respect, and empathy isn't just important for short-term growth, it's essential for long-term business sustainability.

Once business models and goals are clearly understood and communicated, any required IT behaviors and values needed to support them can be influenced through the development of fully aligned IT goals and metrics. These could include shortening lead times for new products to support faster time-to-market, increasing mobile app customer conversions to support increased revenue objectives, or helping the current business scale by making better use of cloud infrastructure.

Lean Thinking to Reduce Waste

Fuel strategies play a significant role in Formula 1 racing. A car with a half-full tank can be as much as three seconds faster than a rival vehicle with a fully loaded fuel cell. Extra fuel equals extra weight, so teams go to great lengths to calculate the exact amount of fuel needed under full race conditions.

Though beautifully engineered and continuously refined, racing car engines are still flagrantly wasteful. Like your car at home, the dynamics of internal combustion still mean that only a certain percentage of fuel stored in the tank is converted into useful energy. The rest is lost as heat and friction and explains why teams constantly refine chassis designs to reduce aerodynamic drag.

Beyond the frenzied world of Formula 1, many elements contribute toward engine waste in the cars we lesser mortals drive. Running an air conditioner consumes fuel without contributing to motion. Friction in engine pistons wastes fuel, as does tire pressure and extra luggage. All told, as little as 14 percent of passenger car fuel is converted into useful energy. Clearly, this gas guzzling engineering throwback is rife for disruption from electric cars and advances in battery technology—time will tell.

In many ways software development is as wasteful as the internal combustion engine. Friction between development and operation causes delays. Manually assembling multiple components and configurations within our own software factories leads to lost time. Carrying excess inventory in the form of unneeded infrastructure capacity adds extra cost. Add to this constraints preventing access to critical systems and data during testing, and defects can accumulate across the software lifecycle.

Lean and Value Creation

The traditional view of IT value has been internally-shaped. For decades, systems and applications have been designed, built, tested, and released to customers, citizens, and end users where they hopefully influenced behaviors. All this has changed. With the advent of cloud, mobility, and social computing, consumers rather than producers call the shots. This means businesses find themselves in the position of having to respond to the behaviors and desires of their customers.

For IT, this redefinition of business value means teams must focus on two essential strategies. First, they must continuously reexamine the software services they deliver from the perspective of the customer, and second, they must constantly strive to minimize any interference or waste across the entire software factory. This includes everything that impedes the flow of value to customers and incurs more cost.

This notion of value being "pulled" by customers and waste elimination is not new. Lean pioneers and practitioners such as Toyota, Motorola, and Xerox redefined manufacturing by applying these principles; understanding that many forms of waste exist across production processes. And, because they add no value to customers, must be clinically removed.

But can Lean principles be applied in IT, where, unlike traditional manufacturing, waste isn't visible across a factory floor through telltale signs like excess physical inventory or idle machinery? Often due to its intangible nature, waste in IT can be hard to identify yet alone eliminate.

Interestingly, the delivery of software bares many similarities to a manufacturing process. In IT, we have the means to respond to value triggers from our customers by quickly designing, developing, and releasing software services. And, since the software delivery lifecycle represents a manufacturing production line within a software factory, we have the guiding context upon which identify all elements of waste that add no value to the business and its customers.

Eight Elements of Waste

As illustrated in Table 3-1, there are eight elements of waste or "Muda" (using Lean terminology) that severely impact the IT group's ability to increase the value of software services.

Table 3-1. Eight Elements of Waste (D.O.W.N.T.I.M.E)

Type of Waste	Examples	Business Outcomes
Defects	Badly designed and poor quality code Non-functional performance issues	Lost customers and revenue; negative brand impact
Overproduction	Delivering features customers don't need or want Procuring extra capacity due to unanticipated performance requirements	Delays, cost over-runs, and budget problems
Waiting	Excessive release backlogs and bottlenecks Infrastructure and data not available for testing Change reviews; security and compliance audits	Slow time-to-market and value; lost opportunities
Non-value added processing	Lengthy problem resolution and fire-fighting Team-based activities prioritized over program-level objectives	Morale issues; high-staff turnover
Transportation	Frequent release rollbacks Development/QA handoffs	Application launch delays; increased cycle times

(*continued*)

Table 3-1. (*continued*)

Type of Waste	Examples	Business Outcomes
Inventory (excess)	Underutilized resources Partially completed work and excessive work in progress	Increased capital and operational costs
Motion	Developers constantly switching Relearning and rework	Lost productivity; talent erosion
Employee knowledge (unused)	Closed retrospectives and stand-up meetings No feedback established from service management (e.g., call center/ service desk)	Missed opportunities to drive improvements

Examining this table it should be noted that there are close relationships and linkages between elements. For example, undetected code *defects* resulting in performance problems may result in an organization purchasing additional hardware capacity, which leads to excess *inventory*, which increases the support burden. In situations like this, waste begets waste and technical debt accumulates to such an extent it becomes difficult to pay off. The result is that essential development is tied up on maintenance and support activities.

Originally coined by Ward Cunningham in the Agile Manifesto, technical debt has tended to be reviewed from a development perspective[4]. After all, if software defects can be identified and eradicated during early stages of development, then production related problems (which could be significantly costlier to fix) can be avoided.

But technical debt can also be created in IT operations. For example, failing to document or visualize business services (and supporting applications an infrastructure) means teams could take longer triaging problems. Here the waste is *non-value added processing*, which again (because of linkages) results in more waste. In this case, increased *transportation* because a release has to be rolled back.

Using the eight elements of waste list, DevOps practitioners can begin a process of identifying waste elements across the software lifecycle. It's important to understand that "toxicity" levels will vary, so mechanisms must be developed to continuously reveal new situations and conditions that can potentially introduce more waste.

[4]The Agile Manifesto: http://www.agilemanifesto.org/

It's also critical not to restrict the exercise to new development. These may become the debt burden of the future, but could only represent a small part of the portfolio. Legacy infrastructure and production applications should also be included because, even though they change less frequency, they often incur significant management costs and overheads.

Finally, debt and the associated waste should be reviewed as a continuum, with special attention paid to integrations between new customer facing apps and essential back-end business processes. For most enterprises, multi-channel engagement creates tremendous opportunity for value creation, but will introduce more waste if they're not integrated and coordinated with existing back-end systems, applications, and call-center services.

Waste Removal Strategies

Today's mobile and API-centric forms of service delivery mean that customers assess value based on extremely high levels of functional and operational quality. They also expect businesses to deliver additional value in the form of continuous change. With customer experience so important, it's critical to begin waste identification from the perspective of the customer; monitoring and analyzing the usage and behaviors of applications and determining what elements impact the total experience. This is especially important for mobile applications, since factors beyond the control of IT departments (e.g., carrier network latency and cloud service performance) can quickly erode value, however good the functional quality.

Some immediate practices cross-functional teams can apply to help identify and eliminate waste, include the following.

Prevent Defects by Removing Constraints

When development and testing is constrained due to lack of access to dependencies (e.g., middleware, web services, and test data), defects can quickly work their way into the code base. This is illustrated in a Service Virtualization survey, which identified that on average, participants require access to 52 dependent elements for development or testing, yet have unrestricted access to only 23 of these.[5]

To circumvent these issues, many development teams often attempt workarounds by hand coding (mocks and stubs), but this doesn't provide for realistic application behavior, causing test validation errors and the late

[5]VOKE Market Snapshot™ Report: Service Virtualization; https://www.ca.com/au/register/forms/collateral/voke-market-snapshot-report-service-virtualization.aspx

discovery of defects. Discussed further in Chapter 5, a more scalable approach is to incorporate Service Virtualization into parallel development and test activities.

Focus on Value to Prevent Overproduction

New application features don't necessarily mean more customer conversations and increased revenue. Unnecessary features can result in additional maintenance overheads and cost. There are many methods DevOps practitioners can use to reduce this form of waste, including:

- Incorporating application experience analytics into monitoring strategies to identify mobile app functions and features that are not used
- Split or A/B testing and funnel or cohort analysis
- Refactoring code elements to reduce complexity, remembering that the cheapest and most reliable components are those that don't exist!

Smoothing Flow to Reduce Wait Times

Like waste element #1, this waste can eventuate due to delays waiting on dependencies during development and testing. In the VOKE report mentioned above, 81 percent of participants identified development delays of waiting for a dependency in order to develop software, reproduce, or fix a defect. Additionally, 84 percent of participants identified QA delays of waiting for a dependency in order to begin testing, start a new test cycle, test a required platform, or to verify a defect.

Caution Never underestimate the wait times associated with accessing test data, since a massive 20 percent of the average software delivery lifecycle is wasted waiting for data, locating it, or creating it manually when none exists.

Excessive wait times may also be due to problems managing highly complex release and deployment processes. However good the code, its ultimate value will be determined by how quickly it can be deployed into production. Manual processes and fragile scripting not only compromise these goals, but also increase the potential for defect code being released. These issues can be addressed by:

- Ensuring all key stakeholders possessing the knowledge to move a service swiftly across lifecycle are involved early and often

- Using smaller batch sizes so that value is delivered to customers at regular intervals
- Developing and automating reusable and repeatable processes to simplify and streamline application releases

Limit Non-Value Added Processing Through Data-Driven Insights

Fixing application problems provides limited value to customers. Rather than wait for problems to occur in a production, IT operations should be involved much earlier in the development lifecycle.

Using tools to share information is especially valuable. By leveraging application performance change impact analysis during a build process, for example, developers can quickly determine any adverse performance conditions their code is introducing.

Reduce Transportation Cost by Automating Deployments

When work is manually handed off from one team to another (e.g., developer to test/QA, QA to operations), critical knowledge can be lost. This could lead to additional delays or the highest transportation cost of all—release rollbacks.

There are many strategies to address these issues, including:

- Reducing the number of handoffs by automating standard tasks and activities
- Ensure release automation tools provide an extensive set of action packs and plug-ins so as to fully deploy at an application level, while also integrating key supporting processes (e.g., configuration management)
- Build more knowledge as releases progress (e.g., establishing application performance management in pre-production)

Eliminate Excess Inventory Across the Software Factory

Minimizing inventory is the hallmark of Lean thinking. As in traditional manufacturing, there are many waste indicators in IT's own software factory. In development, partially completed work can become obsolete before it finds its way into production and should be exposed to ensure it doesn't degrade or corrupt the code base. In operations, excess on-premise server infrastructure acquired as a fail-safe to address unanticipated performance problems could be avoided by establishing monitoring in pre-production.

■ **Note** Costs can accumulate substantially when agile teams acquire specialist tools. Work collaboratively to assess whether the additional cost (training or support) offsets the value delivered to one team.

Prevent Unnecessary Motion with Parallel Development

While transportation waste is associated with the unnecessary movement of software, motion waste involves the unnecessary movement of people. A good example is task switching, where an API developer might shift focus to a new project rather than wait for testing dependencies to become available.

Apart from adding more waste (e.g., delays), task switching can introduce many more problems, especially related to the productivity of developers due to constant interruptions.

Some simple strategies to reduce this waste, especially task switching, include:

- Try to ensure teams have all of the knowledge, tools, and data needed to complete their assigned work
- Simulate and virtualize all dependencies so that development teams can code and test in parallel
- Since as much as 50 percent of testing is wasted by teams trying to locate test data or create it manually, consider supplementing constraint-removal strategies with test data management (see Chapter 5)
- Aim to eliminate unimportant work, meetings, and interruptions. If it isn't delivering value, ask why your team is doing it!

Incorporate Employee Knowledge Using Feedback Loops

While the feedback of production information is important to drive software improvements and improve supportability, it isn't the only place where knowledge can be transferred.

Service desks and call-center processes should also include mechanisms to deliver (to development) important information gained from customers on their usage and response to new application features and functions. Knowledge transfer should also be bi-directional. For example, application experience analytics could (when integrated with incident management processes or even social media) become an early warning mechanism to trigger coordinated responses in the event of mobile app usage problems.

DevOps Metrics

With any IT-driven methodology or program, measuring the effectiveness in a business context is critical. But since DevOps isn't a formal framework, organizations have little guidance in determining what metrics should be used.

This can be problematic and lead to a number of suboptimal practices:

- *Efficiency status-quo*—The IT team falls back to metrics traditionally used to demonstrate technical proficiency in meeting stability and resilience goals. Although these are not necessarily wrong, DevOps metrics should also demonstrate how new processes and automated technologies are impacting the business—for example by speeding time-to-market and reducing lead times.
- *Outputs over outcomes*—Organizations gravitate to metrics that are commonly used in assessing team-level productivity. These can include output-based metrics like number of features delivered or servers provisioned. Metrics in this class can be counterproductive unless balanced with outcome-centric indicators that show results achieved against desired quality levels.
- *Low-hanging fruit*—Organizations select metrics that are easily obtained but not necessarily useful. Since DevOps success is predicated on cultural change, businesses must also measure what's harder to determine but potentially more valuable—namely, how the adoption of DevOps behaviors and values at an organizational level is impacting the business.

Anti-Pattern Metrics

Before embarking on a metrics refresh, organizations should consider all existing measures and their applicability in a DevOps context. Particular attention should be given to carefully review those metrics and incentives that are counter to DevOps principles, as illustrated in Table 3-2.

Table 3-2. Problematic Metric Classes

Metric Class	Examples	Adverse Effects
Vanity Metrics	Lines of code produced Function points created	May be counterproductive since they reward the wrong types of behavior—especially if incentives are linked to the metric. Producing more code and features without validation can inhibit other valuable activities such as refactoring and design simplification.
Intra-Team Metrics	Agile team leaderboards Deployments/ changes prevented	Beware of metrics that pit-teams against each other and use vanity metrics as scoring mechanisms. Strike a balance with metrics and rewards that influence positive inter-team behaviors—such as code sharing, peer reviews, and mentoring. Pay particular attention to metrics that promote an anti-DevOps culture, such as rating operational effectiveness on the ability to prevent releases and deployments.
Traditional Metrics	Mean-time-between-failure (MTBF) FTEs: Servers	With faster delivery of services, some failure is to be expected. Always consider that improving responsiveness can be more important (and less costly) than trying to prevent failures.

Suitability Checklist

When reviewing and developing DevOps metrics, it's also important to consider each against a general suitability checklist:

- *Obtainable*—Culture and behavioral improvements are important to measure, but metrics may be difficult to obtain or quantify. Seek out other related data points to help expose —e.g., staff retention rates/transfers as an indicator of employee morale.
- *Reviewable*—Every metric must stand up to rigorous scrutiny in a business context. Carefully review metrics that can be easily collected, but add no tangible value—e.g., lines of code produced per developer.
- *Incorruptible*—Determine whether each metric can be influenced by team and employee bias. Seek out any associated incentives that can work against a collaborative DevOps culture—e.g., existing SLA bonuses inhibiting change.

- *Actionable*—Any metric must support improved decision making. Exposing A/B testing results can for example be a valuable way to quickly determine the effectiveness of new functionality.

Wherever possible, metrics should also be shareable and have relevance across the software lifecycle to both development and operations. For example, generating security scores at a cross-functional team and divisional level can be used to inform teams about the risks of their actions.

Metrics that Matter

Having determined what not to measure, the next stage is to develop a candidate list of metrics supporting the DevOps program. One common mistake is to measure too many elements, falling back to what's easily collectable. Additionally, metrics applicable to DevOps may be new to organizations (e.g., the speed of deployment, rate of change, and customer responsiveness), so it's important to think broadly how changes to work practices, process and technology can support these goals.

- *People*—Staff related metrics can be the most difficult to collect but are still powerful change indicators. Strong consideration should be given to internal metrics like staff retention rates and training, together with mentoring and knowledge building (e.g., open source contributions and wiki development).
- *Process*—It's important to consider how existing practices will help or hinder new targets being achieved, paying special attention to existing bottlenecks (e.g., security audits only conducted after testing will impact deployment rates).
- *Technology*—Good metrics are those that help teams drive improvements, even after failures (e.g., what is the percentage of failed releases and what percentage of these were due to code defects, manual processing, configuration errors, etc.).

When developing metrics, it's important to maintain balance. Defaulting to metrics skewed toward one particular area (e.g., operational or development efficiencies) can have a negative effect in terms of behavioral improvement

Figure 3-1 illustrates four dimensions and sample metrics that can be used to measure the effectiveness of a DevOps initiative.

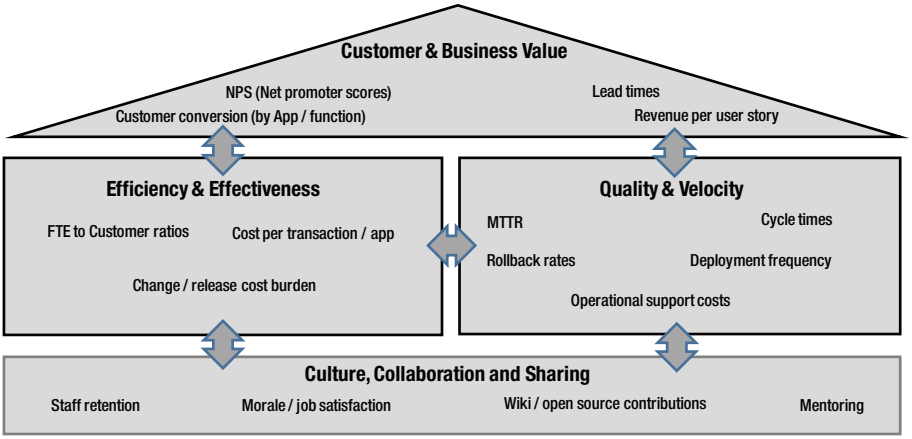

Figure 3-1. DevOps metrics dimensions

Culture, Collaboration, and Sharing

Metrics in this category are especially valuable because they provide an ongoing indicator of acceptance/resistance to DevOps. Some metrics in this dimension will be easier to collect (e.g., staff retention rates/turnover) than others (e.g., employee morale). It's important therefore to look at measures across other dimensions to understand how they impact this area. For example, are mean-time-to-recover (MTTR) improvements positively impacting staff morale, absenteeism rates, and responsiveness to change? Consideration may also be given to automated surveys and employee feedback, as long as these are fully transparent and actionable.

Efficiency and Effectiveness

Metrics here normally focus on elements of development capacity and operational capabilities. While traditional metrics such as server to sysadmin ratios have been used, many organizations are now adopting more customer-centric ratios like full-time-equivalent (FTE) to customers.

Examining full costs on a transactional or application basis is another good candidate metric, as it's focused on improving data center efficiencies (e.g., energy and cooling). Other metrics such as cost of release are also good since these can expose inefficiencies associated with acquiring, preparing, and maintaining physical infrastructure for development, testing, and production.

Quality and Velocity

This dimension looks to measure data points with respect to service delivery. For organizations starting on a DevOps initiative, many indicators (e.g., percentage of deployments rolled-back due to code defects/outages/negative user reactions) could initially be high. This may be a result of the extra time needed to adopt new processes, combined with remediating existing technical debt and waste elements these metrics expose. However, with DevOps' focus on establishing quality right from the start of development, this should reduce over time.

When paired, these metrics also provide additional insights. For example, if the *rate of rollbacks* still increases during periods of low *change volume* it could be indicative of serious problems, e.g., errors due to manual/scripted release processes, task switching, and excessive handoffs.

Other useful metrics in this dimension include:

- *Cycle time*—Measures the length of time it takes to complete a stage or series of stages in a release operation. This can be extremely valuable in exposing any bottlenecks.
- *MTTR*—This can be broken down into detection, diagnosis, and recover phases. MTTR is a great indicator of how effective teams are in handling changes. For complex deployments, there will be spikes, but this metric should be trending down as DevOps becomes established.

Customer and Business Value

This category of metrics are externally focused and help measure how DevOps supports business goals—like increased customer loyalty and faster time-to-market. The manufacturing concept of lead time provides DevOps practitioners with an analogous metric (time taken from when code starts development to successful production deployment) and determines how well DevOps is at meeting the need for rapid delivery of high-quality software services. This metric is especially important to scrutinize because long lead times could be indicative of code defects or testing constraints.

Another interesting candidate is Net Promoter Score (NPS), which is a simple management method to measure customer loyalty. While this metric has traditionally been used in other areas of the business (e.g., marketing), its inclusion is valid since the loyalty of customers is increasingly determined by how quickly high-quality software services and updates can be delivered to via web sites and/or mobile apps.

Additional Methods and Techniques

With metrics developed across each of the four dimensions discussed previously, teams can begin a process of determining the relationships between them. This is important so teams gain insight into what processes enhancements and tools are needed to meet targets or address capability gaps.

One simple and effective approach, as illustrated in Figure 3-2, is business impact mapping. This involves determining which DevOps processes will be needed to support a business or customer experience goal, together with the underpinning metrics, targets, and initiatives/tools across multiple dimensions that support this outcome.

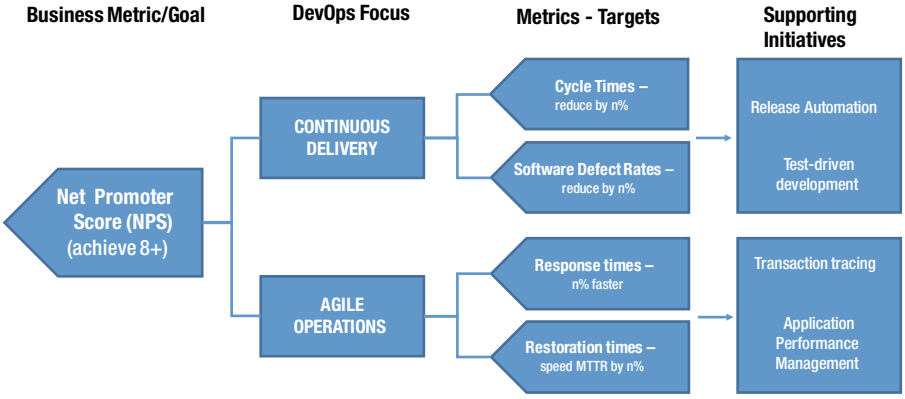

Figure 3-2. Metrics, targets, and initiatives linked to business outcomes

Figure 3-2 illustrates that an organization is seeking to achieve a Net Promoter Score of 8+. To support this goal, IT needs to deliver software releases and new functionality faster, together with ensure a high-quality customer experience. Metrics and targets have therefore been set within the quality and velocity dimension, together with targets and supporting process/tool initiatives.

As DevOps metrics programs develop, practitioners should also:

- Have regular and ongoing target reviews to ensure that goals are not completely unrealistic or that existing processes and tools are not delivering improvements.
- Consider removing persistent "green light" metrics when targets have been consistently achieved.
- Avoid having every metric focused on velocity without paying attention to customer satisfaction and loyalty.

- Strive to prevent vanity metrics and operational or team-centric bias creeping back into the program and distorting the true performance picture.
- Beware of ranking teams based on targets—the best way to compare teams is to measure things like customer loyalty (as described) and how successful teams are in meeting their commitments.
- Give strong consideration to metrics, targets, and initiatives that foster peer review and openness.
- Carefully build incentives and reward programs that reinforce the value of a strong collaborative culture.
- Involve business counterparts right from the start to ensure customer and business data is held to the same standard as operational/efficiency metrics.
- Match tools to the DevOps program, especially those that can monitor and respond to real-time conditions (such as transaction times, response times, and mobile app crashes), but can also proactively detect and prevent adverse conditions (such as code defects and release bottlenecks) that impact performance.

Summary

At its heart, DevOps is about building a generative organizational culture where business improvement is placed above everything else. But as this chapter has illustrated that won't always be straightforward, especially in organizations beset by divisional friction and lack of direction. By leveraging this chapter's guidance, especially with regard to building high-trust teams, an outcome-based metrics program and Lean thinking, organizations have a solid foundation upon which to guide their DevOps programs.

In Chapters 4-7, we'll look closely at the automated tooling needed to support this goal and how businesses can refit and re-engineer their own software factories to manufacture high-quality software innovations, at speed. In these chapters, we'll examine critical tooling strategies across the software lifecycle continuum—Build, Test, Deploy, and Manage.